algorithms

MDPI

Article

A Method for Synthesizing Ultra-Large-Scale Clock Trees

Ziheng Li [1], Benyuan Chen [1,*], Wanting Wang [1], Hui Lv [2], Qinghua Lv [1], Jie Chen [1], Yan Wang [1], Juan Li [1] and Cheng Zhang [1]

[1] National "111 Research Center" Microelectronics and Integrated Circuits, School of Science, Hubei University of Technology, Wuhan 430068, China; zihengl324@gmail.com (Z.L.); 18372832780@139.com (W.W.); linsa080@126.com (Q.L.); 13329944007@163.com (J.C.); hechu748@gmail.com (Y.W.); 2111151135@hbut.edu.cn (J.L.); zhcheng5@163.com (C.Z.)

[2] School of Physics and Electromechanical Engineering, Hubei University of Education, Wuhan 430205, China; simonlv76@gmail.com

* Correspondence: shingolala@126.com

Abstract: As integrated circuit technology continues to advance, clock tree synthesis has become increasingly significant in the design of ultra-large-scale integrated circuits. Traditional clock tree synthesis methods often face challenges such as insufficient computational resources and buffer fan-out limitations when dealing with ultra-large-scale clock trees. To address this issue, this paper proposes an improved clock tree synthesis algorithm called incomplete balanced KSR (IB-KSR). Building upon the KSR algorithm, this proposed algorithm efficiently reduces the consumption of computational resources and constrains the fan-out of each buffer by incorporating incomplete minimum spanning tree (IMST) technology and a clustering strategy grounded in Balanced Split. In experiments, the IB-KSR algorithm was compared with the GSR algorithm. The results indicated that IB-KSR reduced the global skew of the clock tree by 43.4% and decreased the average latency by 34.3%. Furthermore, during program execution, IB-KSR maintained low computational resource consumption.

Keywords: clock tree synthesis; clustering algorithm; k-nearest; VLSI

check for updates

Academic Editor: Arslan Munir

Received: 6 February 2025
Revised: 1 April 2025
Accepted: 18 April 2025
Published: 25 April 2025

Citation: Li, Z.; Chen, B.; Wang, W.; Lv, H.; Lv, Q.; Chen, J.; Wang, Y.; Li, J.; Zhang, C. A Method for Synthesizing Ultra-Large-Scale Clock Trees. *Algorithms* **2025**, *18*, 249. https://doi.org/10.3390/a18050249

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

With the continuous advancement of integrated circuit technology, the complexity of ultra-large-scale integrated circuit design has increased significantly, particularly in modern system-on-chip designs containing millions or even billions of logic elements. The importance of clock distribution networks has become increasingly prominent. As a pivotal component in digital circuit design, the clock network not only determines the synchronization performance of the system but also directly influences power consumption, area utilization and the difficulty of timing closure. The primary objective of the clock tree network is to efficiently distribute clock signals to all clock sinks (such as registers and flip-flops) within the chip while minimizing clock skew and power consumption to ensure the timing performance of the chip. However, with the rapid increase in design size and complexity, traditional clock tree synthesis methods, despite their good performance in small to medium-sized designs, face significant resource bottlenecks when dealing with ultra-large-scale clock networks. Taking the K-means-based register clustering (KMR) algorithm [1] as an example, this is a clock tree synthesis method based on the clustering concept, aiming to reduce global clock skew and power consumption by optimizing the geometric distribution of register clusters. However, when the number of registers reaches the order of hundreds of thousands, the memory and computational resource requirements

faced by the KMR algorithm become exceptionally large. For instance, in a design with 100,000 registers, even storing a minimal spanning tree (MST) of 100,000 simple data points with a basic topological structure requires at least 20 GB of memory—a conservative estimate based on its linear scaling properties. In contrast, the KMR algorithm involves significantly higher complexity, particularly in nanoscale register architectures where each node necessitates extended metadata storage (e.g., 64-bit precision coordinates, timing constraints, and hierarchical dependencies). Experimental data combined with theoretical complexity models reveal that KMR's exhaustive node traversal mechanism, driven by its quadratic $O(n^2)$ scaling behavior, can escalate memory usage to terabyte (TB) levels in large-scale implementations, exceeding the capacity of standard hardware platforms (e.g., laptops with 32 GB RAM). Furthermore, the KMR method is often sensitive to the distance between clusters when optimizing register clusters, potentially leading to higher latency and additional dynamic power consumption in the global clock network. Additionally, other traditional methods, such as the Deferred-Merge Embedding (DME) algorithm, rely heavily on the insertion of buffers to reduce clock skew. However, the excessive use of buffers not only increases power consumption but also imposes additional pressure on the chip's area and physical routing.

The DME algorithm was independently proposed by several groups, including Boese and Kahng, Chao et al. and Edahiro [2]. The DME algorithm is a variant of the Zero-Skew Tree (ZST) algorithm, and it is therefore also referred to as the ZST/DME algorithm. It always yields exact zero skew trees with respect to the appropriate delay model [3]. ZST is not a good choice in practice owing to its high cost, which led to the emergence of the Bounded-Skew Tree (BST). The ZST/DME is extended to the BST/DME by generalizing the merging segments to regions [4]. The BST/DME can produce a set of routing solutions with smooth skew and wirelength tradeoffs [5]. Chen et al. [6,7] equated the clock tree synthesis problem to the problem of constructing a shallow-light tree and proposed an effective algorithm for building a Steiner shallow-light tree while balancing between shallowness and lightness. They proved the equivalence between the wirelength minimization of the ZST and the diameter sum minimization of hierarchical clustering, and they proposed better algorithms for both the ZST and BST [6,7]. Li et al. [8] introduced the skew-latency-load tree, which combines the merits of the BST and Steiner shallow-light tree. In addition, they provided a hierarchical CTS framework, and it is constructed by integrating partition schemes and buffering optimization techniques [8]. Lerner et al. [9] present the definition of bounded slew merging regions that are a conceptual shift in satisfying slew constraints during CTS. This major conceptual shift extends the popular DME framework of the CTS literature for ease of adoption by proposing the novel SMRcts algorithm with slew merging regions [9]. In addition to DME-based algorithms, register-clustering-based algorithms represent another branch of thought in addressing the problem of clock tree synthesis. Wu et al. [10] propose a modified K-means algorithm which effectively assigns flops into clusters at the clustering step. Then, at the relocation step, flops are actually relocated, and regularly structured clusters are formed. Han et al. [11] propose a dynamic programming-based method to determine optimal clock power, skew and latency in the space of generalized H-tree solutions. And they further propose a balanced K-means clustering and a linear programming-guided buffer placement approach to embed the generalized H-tree with respect to a given sink placement [11]. Wang et al. [12] proposed a clock tree synthesis scheme based on flexible H-tree. To reduce the magnitude of the clock-induced on-chip variation, Mangiras et al. [13] incrementally relocate the flip-flops and the clock gaters in a bottom–up manner to implicitly guide the clock tree synthesis engine to produce clock trees with increased common clock tree paths. Kundu et al. [14] have presented an unsupervised machine learning-based multi-bit flip-flop clustering and relocation frame-

work to address the clock network power reduction without impacting the performance of the design. Deng et al. [1] have improved the KMR algorithm and then proposed the K-splitting-based register clustering (KSR) algorithm. By introducing a register clustering strategy, KSR optimizes register distribution and reduces power consumption and latency in the clock tree network, achieving satisfactory results. By effectively clustering registers, KSR groups geographically close registers together, thereby reducing the total delay in clock distribution and lowering power consumption. Despite having excellent performance in power consumption optimization, KSR still faces numerous challenges when dealing with ultra-large-scale clock tree synthesis, particularly in designs with over 100,000 registers where it only stores the minimum spanning results. These challenges primarily stem from bottlenecks in computational resources and memory.

Therefore, in response to the limitations of the KSR method in ultra-large-scale clock tree synthesis, this paper proposes an improved clock tree synthesis method, named IB-KSR. The IB-KSR method optimizes upon the traditional KSR method, aiming to effectively address the computational resource and memory constraints in ultra-large-scale clock tree synthesis while further enhancing the overall performance of the clock tree network. By improving the register clustering strategy and optimizing the clustering scale, IB-KSR is able to achieve efficient clock distribution in larger-scale designs while striking a better balance between timing and resource consumption. This approach brings fresh perspectives to tackle the challenges of ultra-large-scale clock tree synthesis and offers a more scalable and efficient solution for future clock network optimization. The authors mainly optimized the construction of the minimum spanning tree (MST) in the KSR algorithm by adopting the incomplete MST (IMST) technique, which significantly reduced the time complexity and space complexity. Specifically, by this optimization, the time complexity and space complexity have been effectively controlled, enhancing the performance of the algorithm in ultra-large-scale clock tree synthesis. Furthermore, this optimization is obviously more suitable for clock tree synthesis with clustering as the primary method, as we will elaborate later. Furthermore, we have introduced a balanced splitting technique to strictly control the size of clustering clusters. This technique enables fine-grained control over the fan-out of each buffer while ensuring a certain level of clock skew and buffer insertion, thereby effectively reducing power consumption and latency. During the experimental stage, the authors comprehensively verified the proposed IB-KSR algorithm using 10 sets of register placement data. Moreover, the authors implemented the greedy search-based register clustering (GSR) algorithm also proposed by Deng [1] and conducted a comparison between the two algorithms. The results demonstrate that in the application of ultra-large-scale clock tree synthesis, the IB-KSR algorithm exhibits significant advantages over the GSR algorithm in controlling clock skew and average delay. To further evaluate the applicability of the IB-KSR algorithm, the authors selected some benchmark test data from ISPD 2010 [15] and conducted a comparison with the KSR algorithm.

The authors compared exclusively with GSR and KSR because all three algorithms were developed by the same research team, sharing a common lineage and continuity. Moreover, since both GSR and KSR are clustering-based clock tree synthesis methods, the comparison with GSR more effectively demonstrates the scalability of clustering algorithms.

Although the ISPD 2010 dataset is somewhat outdated, the purpose of this experiment is not to validate clock tree synthesis performance based on this dataset. Instead, it aims to compare the differences between the improved IB-KSR and KSR. Since it is difficult to apply KSR to large-scale datasets in our experimental environment, we chose the ISPD 2010 dataset. Moreover, numerous publicly available experimental results are based on this benchmark, allowing for direct comparisons.

The results indicate that IB-KSR performs similarly to KSR in clock tree synthesis for medium to small-scale designs. However, IB-KSR has lower memory usage and higher computational efficiency. And as the scale increases, its unique IMST and Balanced Split mechanisms come into full play, demonstrating significant performance advantages. The results of both comparisons have verified the superiority and broad applicability of the IB-KSR algorithm in clock tree synthesis across different scales.

2. IB-KSR Algorithm

2.1. IMST

Traditional MST generation methods include algorithms such as Kruskal and Prim. Regardless of the algorithm used, they all need to traverse all nodes and store a significant amount of edge weight information. Taking Kruskal as an example, for node information involving 100,000 nodes, merely using the int type in C for storage would require approximately 20 GB of memory (considering additional data structures and overhead, this estimate can vary, but it illustrates the scale of the requirement). Moreover, the running time of such an approach would be prohibitively long. Thus, we employed the IMST technique, which does not compute and store all edges but only stores the nearby edges with smaller weights.

As shown in Figure 1, the specific content is to traverse all nodes, using each node as a starting point for scanning, scan the k nearest nodes to that point, and generate k sets of edge information. In essence, this part constitutes a k-nearest neighbor method.

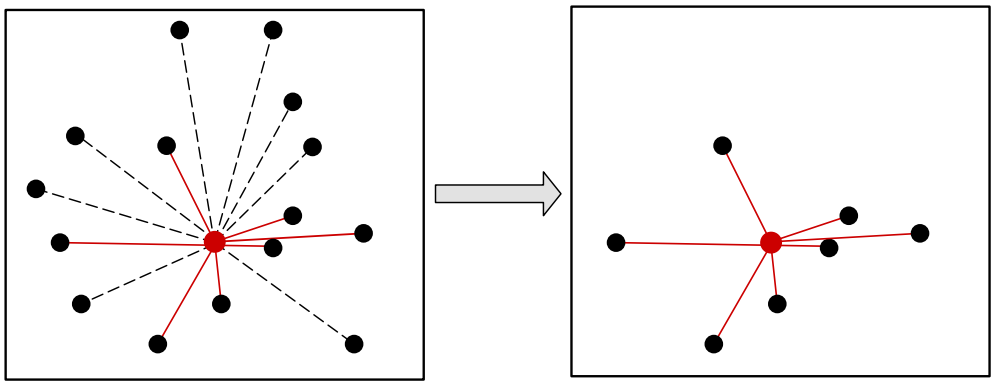


Figure 1. With $k = 7$, the dashed lines represent the edges that Kruskal's algorithm would need to compute, while the red lines depict the k sets of edges scanned using the k-nearest neighbors method. Note that the selection of nearest neighbors is based on Manhattan distance rather than the intuitive Euclidean distance. The red circle serves as the reference point for the k-nearest neighbors search, where the algorithm identifies the k closest data points to this centroid. The black circle represent candidate points in the search space.

The complete description of the IMST algorithm is presented as Algorithm 1 below:

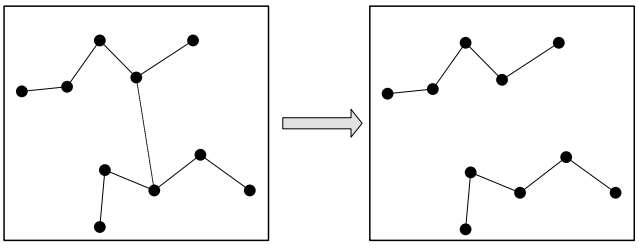
For n nodes, this approach generates a total of kn sets of edge information, which are added to a set E to form an incomplete edge set E_k. After sorting E_k in ascending order, Kruskal's edge addition method is applied to construct an incomplete minimum spanning tree. In the process of finding the nearest neighbor nodes, there are indeed many methods that can achieve logarithmic time complexity $O(\log n)$ for the search. One such method is the use of R-trees found in C++'s Boost library. However, R-trees are based on Euclidean distance for nearest neighbor search, but in the context of clock tree synthesis, the Manhattan distance better aligns with the actual physical constraints in a circuit layout. Thus, we adopted a quadtree-based nearest neighbor search technique. Quadtrees, as a type of spatial partition data structure, efficiently support k-nearest neighbor searches

based on Manhattan distance by recursively dividing the plane into subregions. When combined with a priority queue algorithm, the search efficiency can be further optimized while ensuring accuracy, resulting in an overall time complexity of $O(k \log n)$, where k is the number of nearest neighbors and n is the number of nodes.

Algorithm 1. Incomplete Minimum Spanning Tree (IMST)

Input: A set of points $P = \{r_1, r_2, \ldots, r_n\}$, with the location (x_{r_i}, y_{r_i}) for point r_i
Output: A minimum spanning tree (IMST) with edges E_{IMSTE}
1: Create a graph $G(P, E)$, where $E = \{e(r_i, r_j) | i, j \in \{1, 2, \ldots, n\} \cap i \neq j\}$
2: **For** each point $r_i \in P$:
3: Create an empty list neighbors to store the nearest neighbors of r_i
4: **For** each of the k nearest neighbors r_j of r_j:
5: Calculate the Manhattan distance between r_j and r_j:
$d(r_i, r_j) = \left|x_{r_i} - x_{r_j}\right| + \left|y_{r_i} - y_{r_j}\right|$
6: Add the edge $e(r_i, r_j, d(r_i, r_j))$ to the list neighbors.
7: **End For**
8: Add these k edges to the edge set E_k
9: **End For**
10: Sort all edges in E_k in ascending order by weight.
11: Apply Kruskal's algorithm to the sorted edge list E_k to generate the incomplete minimum spanning tree (IMST)
12: Return the set of edges E_{IMST}, which forms the IMST of the point network

Since IMST only considers the k nearest neighbor edge information for each node and omits the computation and storage of distant edges, the resulting structure not only saved computational resources but also naturally adapted to the requirements of clock tree synthesis. However, this incomplete edge information also results in the output structure of IMST differing from that of an MST. Specifically, the result of IMST is a forest consisting of multiple trees rather than a single complete tree, as shown in Figure 2.



(a) MST (b) IMST

Figure 2. (**a**) The tree obtained using the MST algorithm, while (**b**) depicts the forest composed of two trees obtained by using the IMST algorithm.

The forest structure of IMST reflects its essential characteristics:

- Natural cutting of long edges: Since each node only establishes connections with its k nearest neighbors, connections between distant nodes are automatically omitted.
- Preliminary clustering: Each independent tree in the forest is composed of nodes that are geographically close to each other, thus naturally forming a preliminary cluster.

These characteristics make IMST technology more advantageous in the context of clock tree synthesis. IMST significantly reduces the number of edges, thereby decreasing memory usage and computational overhead. Moreover, in the forest generated by IMST, each tree naturally forms a cluster of neighboring nodes, providing an efficient starting

point for subsequent clustering steps (such as buffer insertion). In short, the improvement of IMST lies in replacing the fully connected initialization of traditional MST with a non-fully connected sparsification: by screening k-nearest neighbors to retain strongly correlated edges. The generated IMST exhibits a topology similar to MST and is more suitable for subsequent clock tree clustering.

2.2. Balanced Split

During clock tree synthesis, buffers have certain fan-out limitations. This limitation, when reflected in the clustering algorithm, means that an upper limit needs to be set for the size (number of nodes) of a single cluster. Many additive clustering algorithms can effectively enforce this constraint. For example, the GSR algorithm starts with an initial cluster and gradually adds nodes to it. When the cluster size reaches the set limit, it is easy to terminate the expansion of the current cluster and create a new cluster to continue adding nodes. This method provides real-time control over the size in an incremental manner, making it very intuitive.

Alternatively, for the KSR algorithm, the clusters are generated by cutting a tree. Specifically, KSR generates multiple clusters by cutting edges to divide a single tree into two subtrees. In KSR, the number of clusters is controlled by a parameter. A threshold edge length is calculated using Equation (1), and the algorithm cuts all edges with weights greater than this threshold to generate the target number k of clusters. In Equation (1), Width and Length are the width and the length of the layout, respectively, $Size_{ob}$ is the total size of the obstacles on the layout, $Num_{registers}$ is the total number of the registers on the layout, and α is a user-defined parameter to control the amount of clusters to be generated [1]. However, this method only constrains the number of clusters rather than their size (number of nodes). Hence, the size of a single cluster is hardly controllable. Experiments have shown that in a clock tree containing 100,000 registers, the maximum cluster size in the initial clustering can reach several thousand nodes. Such excessively large clusters can cause significant pressure on subsequent clustering steps, making it difficult to meet the requirements of buffer fan-out limitations.

$$EL = \alpha \times \sqrt{\frac{Width \times Length - Size_{ob}}{Num_{registers}}} \tag{1}$$

In order to control cluster size, skew, and bus length simultaneously, the authors propose a balanced cutting technique. This technique is based on the principle that for any tree, there must exist an edge, cutting which divides the tree into two parts with the minimum size difference between them. We refer to this edge as the balancing edge. By cutting the balancing edge, the two resulting subtrees not only have the minimum size difference but also significantly reduce the overall size of the original tree. This method ensures size control while avoiding unnecessary skew and bus length increases due to excessive splitting.

The implementation of this algorithm can be divided into two main steps. Firstly, a Breadth-First Search (BFS) is conducted starting from an arbitrary node to traverse the entire tree so as to update the parent node information for each node. This computational step is formalized as Algorithm 2, with its pseudocode structure detailed below.

Next, starting from an arbitrary leaf node, using the parent node information obtained in the previous step, a reverse traversal is conducted along the structure of the tree. During this process, the subtree size of each node is progressively updated using a prefix sum approach with the subtree size of leaf nodes being initialized to 1. After updating the sizes of the subtrees, a forward traversal is conducted once more to calculate the size difference between the left and right subtrees based on Equation (2). Subsequently, the node with the

smallest discrepancy is selected as the balancing node. Eventually, the edge pointing from this node to its parent is identified as the balancing edge, thus achieving a balanced cut of the tree. This step is formalized as Algorithm 3 and described as follows.

$$diff(u) = \left| \frac{n}{2} - SubtreeSize[u] \right| \tag{2}$$

Algorithm 2. Parent Node Update

Input: A tree represented by an edge set $E = \{e(u, v)|u, v \in V\}$, where V is the set of nodes.
Output: An array $Parent[]$ for each node u, where $Parent[u]$ represents the parent of node u in the tree.
1: Initialize $Parent[] =$ NULL for all $u \in V$.
2: Select an arbitrary node $r \in V$ as the root of the tree.
3: Create an empty queue Q.
4: Enqueue the root r into Q.
5: **While** Q is not empty:
6: Dequeue a node u from Q.
7: **For** each adjacent node v *of* u such that $Parent[v] =$ NULL
8: Set $Parent[v] = u$.
9: Enqueue v into Q.
10: **End For**
11: **End While**
12: Return the $Parent[]$ array.

Algorithm 3. Balanced Edge identification

Input: A tree represented by an edge set $E = \{e(u, v)|u, v \in V\}$ and the $Parent[]$ array obtained from the Parent Node Update algorithm.
Output: A balanced edge $e_b = (u, Parent[u])$, which minimizes the size difference between the two subtrees when the edge is cut.
1: Let $n = |V|$ be the number of nodes in the tree.
2: **For** each node $u \in V$, Initialize $SubtreeSize[u] = 1$. **End For**
3: Let $Leaves = \{u \in V|\text{degree(u)} = 1 \cap u \neq r\}$, which is the set of leaf nodes.
4: **While** $Leaves \neq 0$:
5: Dequeue node u from $Leaves$.
6: Let $p = Parent[u]$, the parent of u.
7: Update $SubtreeSize[p] += SubtreeSize[u]$.
8: Remove edge $e(u, p)$.
9: If p becomes a leaf, add p to $Leaves$.
10: **End While**
11: Initialize $diff_{\min} = \infty, e_b =$ NULL.
12: **For** each node $u \in V$:
13: Compute $diff(u)$ with Equation(2).
14: If $diff(u) < diff_{\min}$, update $diff_{\min}$ and $e_b = (u, Parent[u])$.
15: **End For**
16: Return the balanced edge e_b.

2.3. Detail of IB-KSR

The IB-KSR algorithm is an improved version based on the KSR algorithm. The algorithm first generates an IMST based on the input set of registers. Then, it calculates the

threshold EL using Equation (1) and prunes the edges in the IMST with weights greater than EL, resulting in an initial tree structure. Next, for trees with degrees exceeding the maximum fan-out of the buffers, a balanced cut operation is performed to generate initial clusters.

For each cluster, the IB-KSR algorithm employs a heuristic Depth-First Search (DFS) algorithm to search for the optimal buffer insertion locations in order to balance the skew and total wire length of the clock tree. The inserted buffers will serve as the starting points for the next layer of clustering. This process continues until all clustering is completed, ultimately forming a complete clock tree. The complete IB-KSR algorithm, designated as Algorithm 4, is outlined as follows.

Algorithm 4. IB-KSR

Input: A set of registers $S_{register} = \{r_1, r_2, \ldots, r_n\}$ with the location (x_{r_i}, y_{r_i}) for register r_i and maxfount.
Output: A complete clock tree.
1: Convert the register information $S_{register} = \{r_1, r_2, \ldots r_n\}$ with locations (x_{r_i}, y_{r_i}) into a set of points $P = \{(x_{r_1}, y_{r_1}), (x_{r_2}, y_{r_2}), \ldots, (x_{r_n}, y_{r_n})\}$.
1: **While** P.size() > 0
2: Generate IMST $E_{IMST} =$ IMST(P). (Algorithm 1)
3: Calculate WL with Equation (1).
4: Cut edges with weight $w(e) > EL$, $E_{IMST} =$ CutEdges(E_{IMST}, EL).
5: Initialize Max-Heap $H =$ Heap$_{\max}(E_{IMST})$
6: **While** H.top.size $> maxfount$:
7: Extract max tree $E_{\max} = H$.pop()
8: Perform Parent Node Update on $E_{\max}$, ParentNodeUpdate$(E_{\max})$. (Algorithm 2)
9: Perform Balanced Edge identification, $e_{balance} =$ BalancedEdgeIdentification$(E_{\max})$. (Algorithm 3)
10: Split $E_{\max}$ at $e_{balance}$, create new two trees E_1, E_2.
11: H.push(E_1, E_2).
12: **End While**
13: Initialize $Points_{next} =$ NULL.
14: **For** each cluster $C \in H$
15: Perform DFS on C to find the optimal buffer insertion node r_b.
16: $r_b.children$.push(C)
17: $Points_{next}$.push(r_b).
18: **End For**
19: Update $Points = Points_{next}$.
20: **End While**
21: Return clock tree $Tree_{clock} =$ Extract$ClockTree(Points)$.

The core idea of the heuristic DFS search algorithm is to prune in unnecessary directions. Specifically, we assume that moving the insertion point in a single direction will monotonically affect the skew of the clock tree. Therefore, during each recursive process, the algorithm will prune those directions that would lead to an increase in skew. The time complexity of the algorithm is $O(\log(n))$, where n is the number of candidate insertion points.

3. Applicability and Complexity of the IB-KSR Algorithm

3.1. Applicability

The IB-KSR algorithm is suitable for very large-scale clock tree synthesis (with >100 *k* nodes). It leverages the initial construction of an IMST to effectively avoid excessive computational resource consumption. The IMST explores the neighborhood topology information in the graph and stores this information in independent tree structures. Using Equation (1) for initial cutting allows neighboring nodes to naturally form clusters. Moreover, Equation (2) is utilized to further control the size of the clusters, effectively alleviating the fan-out pressure on the buffers.

The IB-KSR algorithm not only endows the original KSR with the capability to handle ultra-large-scale clock tree synthesis but also retains the performance of the original KSR in the synthesis of small- to medium-scale clock trees (<1 k nodes). When the number of registers is relatively small, the structures generated by the IMST technique tend to closely resemble complete minimum spanning trees, causing the algorithm's behavior to gradually converge to that of the original KSR algorithm. The subsequent evaluation and validation phases will also demonstrate that the IB-KSR algorithm will exhibit significant advantages in ultra-large-scale clock tree synthesis while maintaining high performance in small- to medium-scale scenarios.

Moreover, for scenarios requiring overlap control, overlap constraints can be incorporated into the DFS process. This effectively manages the overlap between buffers and registers, thereby enhancing the overall design quality. The IB-KSR algorithm demonstrates strong versatility, adapting to clock tree synthesis tasks of varying scales while delivering optimized results.

3.2. Complexity

The IB-KSR algorithm primarily consists of five steps. Assuming the dataset contains n registers, the following derivation will be presented according to these five steps.

Step 1: Constructing the initial connected graph (IMST technique): The initial connected graph is constructed using the IMST technique, with a time complexity of $O(k \cdot n \cdot log\ n)$, where k represents the predefined number of nearest neighbors.

Step 2: Edge cutting: Edge cutting is performed according to Equation (2). This step utilizes a max heap to manage edges that exceed the threshold. Let m_1 represent the number of edges exceeding the threshold (where $m_1 << n$). Since each heap operation on an edge requires $O(log\ m_1)$ time, the time complexity of this step can be estimated as $O(m_1 \cdot log\ m_1)$.

Step 3: Balanced cutting (for trees exceeding the maximum fan-out limit, assuming the total number of edges is m_2, where $m_2 < n$): First, the nodes of these trees need to be traversed twice, requiring $O(2m_2)$. Then, balanced cutting is performed on each subtree that exceeds the limit (let the number of such subtrees be m_3, where $m_3 << n$). Each cutting operation involves an insertion operation in the max heap, taking $O(log\ m_3)$ time. Assuming all relevant edges need to be processed, the time complexity of this part is approximately $O(m_2 \cdot log\ m_3)$. Therefore, the overall time complexity of Step 3 is approximately $O(m_2 + m_2 \cdot log\ m_3)$, which can be simplified to $O(m_2 \cdot log\ m_3)$ when constants are ignored.

Step 4: Buffer insertion point search: Each tree generated after balanced cutting is treated as a cluster. The search for buffer insertion points within each cluster is conducted using unidirectional pruning and overlap pruning techniques. Let the number of feasible candidate points in each cluster be m_4 (where $m_4 \approx n$), and let the number of clusters be m_5 (where $m_5 << n$). The time complexity of this step is $O(m_5 \cdot log\ m_4)$.

Step 5: Iterative merging: All newly inserted buffers are treated as the starting points for the next iteration of Step 1, and the above process is repeated until only one point remains directly connected to the clock source. The entire process requires $O(log\ n)$ iterations.

In summary, the overall time complexity can be estimated as $O(\log n \cdot [k \cdot n \cdot \log n + m_1 \cdot \log m_1 + m_2 \cdot \log m_3 + m_5 \cdot \log m_4])$. Given that $m_1 << n$, $m_2 < n$, $m_3 << n$, $m_4 \approx n$ and $m_5 << n$, the primary time cost is concentrated in the $O(k \cdot n \cdot log\ n)$ term. After considering the iterations, the overall complexity is approximately $O\left(k \cdot n \cdot (log\ n)^2\right)$. In comparison, the time complexity of KSR in the simplest case can be as high as $O\left(n^2 \log(n)^2\right)$.

The space complexity calculation is relatively straightforward. It primarily consists of the space required to store the graph, while other data structures occupy minimal space due to the use of pointer operations. The space complexity of the IB-KSR algorithm is approximately $O(k \cdot n)$, whereas the KSR algorithm requires up to $O\left(n^2\right)$.

4. Experiment

4.1. Comparison Between IB-KSR and GSR

The authors utilized 10 sets of register placement data provided by the 2024 EDA Elite Challenge and implemented Deng's GSR algorithm to conduct a comparison with the IB-KSR algorithm in addressing ultra-large-scale clock tree synthesis. The testing platform was equipped with an Intel Core i7-11800 processor and 40 GB of memory.

For the ten sets of register placement data, the IB-KSR algorithm demonstrated exceptional performance, as shown in Table 1. Compared to the GSR algorithm, the IB-KSR algorithm achieved a 43.4% improvement in global skew and a 34.3% improvement in average clock latency. This significant improvement came at the cost of only a 21.4% increase in the number of buffers.

Another critical challenge in large-scale clock tree synthesis is the overlap issue between buffers and registers as well as between buffers themselves. To address this problem, the authors introduced an overlap detection mechanism during buffer insertion. Although IB-KSR and GSR share the same overlap detection strategy, GSR's structural characteristics limit the available selection space for the next-level nodes within a single clustering cluster. This limitation may make it difficult to find positions within the entire cluster where no overlap occurs. In contrast, IB-KSR significantly reduces the overlap rate by its tree-like distributed clustering strategy, demonstrating a 95.6% advantage in this metric.

It should be noted that the performance improvement of the IB-KSR algorithm comes at the cost of increased program runtime with its synthesis time being approximately 87 times that of GSR. However, since the optimized algorithm can complete the synthesis within five minutes, the time consumption is still within an acceptable range.

The primary reason for selecting GSR as a comparison target is that GSR and KSR share a common origin and both belong to clustering-based clock tree synthesis algorithms, which allows for a better demonstration of the improvements and innovations of IB-KSR within the same category of algorithms. Additionally, although GSR has existed for many years, it is considered a classic benchmark algorithm, and no direct improvements based on GSR for clock tree synthesis have been proposed to date. Therefore, GSR remains a meaningful point of comparison. Consequently, conducting performance comparisons between GSR and IB-KSR implementations in clock tree synthesis constitutes a technically valid and operationally meaningful evaluation framework.

Table 1. Comparison between IB-KSR and GSR.

Benchmark	Register	Algorithm	Global Skew (ps)	Average Clock Latency (ps)	Buffer Count	Overlap Area	Total Time (s)
Case1	121949	IB-KSR	145.0863	493.1913	3167	0.43%	252.33
		GSR	231.6535	827.4057	2597	6.88%	2.93
Case2	114261	IB-KSR	102.2597	518.1393	2970	0.31%	158.46
		GSR	114.3043	526.4502	2651	3.99%	2.11
Case3	139387	IB-KSR	112.2617	472.9788	3639	0.25%	251.36
		GSR	158.3193	742.1908	3167	7.26%	3.02
Case4	137783	IB-KSR	138.8413	631.1416	3549	0.29%	268.63
		GSR	363.7603	887.4536	2906	6.14%	3.25
Case5	184204	IB-KSR	152.9521	595.4133	4788	0.28%	331.92
		GSR	319.6448	972.6395	4132	6.68%	3.59
Case6	178852	IB-KSR	154.3807	519.8054	4657	0.33%	390.40
		GSR	152.6932	797.166	3940	7.99%	4.01
Case7	144441	IB-KSR	96.6709	552.9979	3822	0.39%	273.92
		GSR	172.7921	473.3328	3296	8.61%	2.88
Case8	138733	IB-KSR	148.5577	479.536	3698	0.18%	288.28
		GSR	217.1041	799.3908	3039	6.62%	3.14
Case9	166093	IB-KSR	117.9646	569.4214	4318	0.2%	354.01
		GSR	434.5138	1211.0952	3519	7.29%	4.21
Case10	184987	IB-KSR	211.6576	622.0583	4968	0.37%	314.48
		GSR	259.0176	919.1639	4072	5.87%	3.98
Average		IB-KSR	137.47	535.75	3953	0.3%	288.78
		GSR	242.88	815.83	3257	6.83%	3.31
Comparison		$\frac{IB\text{-}KSR}{GSR}$	0.57	0.66	1.21	0.04	87.24

4.2. Comparison Between IB-KSR and KSR

To validate the performance of the IB-KSR algorithm in small- to medium-scale scenarios, the authors conducted a comparison with the KSR algorithm using a portion of benchmark data from ISPD 2010. Since the KSR algorithm is not suitable for ultra-large-scale clock tree synthesis, IB-KSR does not activate the IMST and Balanced Split modules in smaller-scale clock trees, and its performance is expected to closely resemble that of the native KSR algorithm. The comparison results are shown in Table 2.

Table 2. Comparison between IB-KSR and KSR on ISPD 2010 benchmark tests.

Benchmark	Register	Algorithm	Global Skew (ps)	Average Clock Latency (ps)	Total Time (s)	Average RSS (MB)	Max RSS (MB)
10f01	1107	IB-KSR	55.5352	436.8898	0.096	5.049	8.520
		KSR	54.5593	459.9714	0.220	20.813	35.212
10f02	2249	IB-KSR	71.4987	616.7572	0.168	7.044	15.208
		KSR	77.1856	677.0919	0.593	100.849	170.880
10f03	1200	IB-KSR	18.8928	191.0488	0.057	7.012	10.540
		KSR	27.7179	191.7802	0.151	30.035	51.144
10f04	1845	IB-KSR	15.1817	216.4092	0.123	7.415	12.496
		KSR	22.7395	203.2573	0.107	80.815	140.544
10f05	1016	IB-KSR	16.7752	130.0987	0.077	6.311	10.564
		KSR	11.5607	125.1863	0.129	13.567	38.14
Average		IB-KSR	35.5767	330.4407	0.104	6.566	11.466
		KSR	38.7526	331.3111	1.20	49.216	87.184
Comparison		$\frac{IB\text{-}KSR}{KSR}$	0.92	0.997	0.087	0.13	0.13

In small- to medium-scale clock tree synthesis, the performance metrics of IB-KSR are close to those of the native KSR algorithm. As the scale of the clock tree increases, the IMST and Balanced Split modules of IB-KSR begin to demonstrate their advantages. In smaller-scale scenarios, however, the performance of IB-KSR gradually converges to that of the KSR algorithm. This indicates that IB-KSR not only holds significant advantages in ultra-large-scale clock tree synthesis but also exhibits broad applicability across clock tree synthesis tasks of varying scales.

Through a comparative analysis of Resident Set Size (RSS) metrics, IB-KSR demonstrates significantly superior memory efficiency that exhibits nonlinear scalability advantages. In large-scale clock tree synthesis scenarios constrained by buffer fan-out limitations, IB-KSR maintains practical nearest-neighbor parameter configurations (k-value < 1 k), while conventional KSR implementations must process exhaustive node traversals-potentially involving millions of nodes. This fundamental architectural difference results in KSR's memory footprint scaling to terabyte-level requirements (1000× greater than IB-KSR). Although terabyte-capacity servers exist, from a cost-performance perspective, KSR proves fundamentally impractical for ultra-large clock tree implementations. Moreover, its runtime latency could extend beyond 10 h (1000× slower), further compromising operational efficiency.

5. Conclusions

To address the challenges of computational resource limitations and buffer fan-out bottlenecks in ultra-large-scale clock tree synthesis, this paper improved and extended the KSR algorithm, proposing the IB-KSR clustering algorithm. This algorithm retains the advantages of low power consumption and low latency inherent in the KSR algorithm while enhancing adaptability to ultra-large-scale clock trees through optimized synthesis processes. It significantly improves computational efficiency and resource utilization.

Experimental results demonstrate that compared to the GSR algorithm, IB-KSR exhibits significant advantages in clock skew control, achieving a 43.4% reduction in maximum skew and a 34.3% improvement in average latency. Despite handling clock trees with hundreds of thousands of nodes, IB-KSR maintains its runtime within a manageable range, fully showcasing its efficiency in ultra-large-scale clock tree synthesis.

In summary, the IB-KSR algorithm not only preserves the superior low power consumption and low latency characteristics of KSR but also introduces a highly efficient and scalable synthesis solution by design optimizations tailored for ultra-large-scale clock trees. By effectively addressing the technical challenges of computational resource limitations and fan-out bottlenecks, the algorithm provides a practical and feasible approach to achieving efficient ultra-large-scale clock tree synthesis. Additionally, the algorithm's unified framework can be integrated as a modular component into existing EDA platforms and supports standard data formats. Its scalability and efficiency make it suitable for both small-scale and large-scale designs, reducing the need for multiple specialized tools.

Author Contributions: Conceptualization, Z.L. and C.Z.; methodology, Z.L.; software, Y.W.; validation, Z.L. and Y.W.; formal analysis, J.C.; investigation, C.Z.; resources, B.C.; data curation, J.L.; writing—original draft preparation, W.W.; writing—review and editing, B.C.; visualization, J.L.; supervision, B.C.; project administration, Q.L.; funding acquisition, H.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the Wuhan Key Research and Development Plan (Grant No. 2024050702030134), China-Sudan Joint Laboratory of New Photovoltaic Ecological Agriculture (Grant No. 2023YFE0126400) and College Student Innovation and Entrepreneurship Training Program (202310500013).

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Deng, C.; Cai, Y.C.; Zhou, Q. Register Clustering Methodology for Low Power Clock Tree Synthesis. *J. Comput. Sci. Technol.* **2015**, *30*, 391–403. [CrossRef]
2. Kahng, A.B.; Lienig, J.; Markov, I.L.; Hu, J. *VLSI Physical Design: From Graph Partitioning to Timing Closure*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2011.
3. Chao, T.H.; Hsu, Y.C.; Ho, J.M.; Kahng, A.B. Zero skew clock routing with minimum wirelength. *IEEE Trans. Circuits Syst. II Analog Digit. Signal Process.* **1992**, *39*, 799–814. [CrossRef] [PubMed]
4. Chen, G.; Young, E.F.Y. Dim Sum: Light Clock Tree by Small Diameter Sum. In Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 25–29 March 2019.
5. Cong, J.; Kahng, A.B.; Koh, C.K.; Tsao, C.W.A. Bounded-Skew Clock and Steiner Routing. *ACM Trans. Des. Autom. Electron. Syst.* **1998**, *3*, 341–388. [CrossRef]
6. Chen, G.; Young, E.F.Y. SALT: Provably Good Routing Topology by a Novel Steiner Shallow-Light Tree Algorithm. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 1217–1230. [CrossRef]
7. Chen, G.J. *VLSI Routing: Seeing Nano Tree in Giga Forest*; The Chinese University of Hong Kong: Hong Kong, China, 2019.
8. Li, W.G.; Huang, Z.P.; Yu, B.; Zhu, W.X.; Li, X.Q. Toward Controllable Hierarchical Clock Tree Synthesis with Skew-Latency-Load Tree. In Proceedings of the 61st ACM/IEEE Design Automation Conference, San Francisco, CA, USA, 23–27 June 2024; pp. 1–6.
9. Lerner, S.; Taskin, B. Slew Merging Region Propagation for Bounded Slew and Skew Clock Tree Synthesis. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1–10. [CrossRef]
10. Wu, G.; Xu, Y.; Wu, D.; Ragupathy, M.; Mo, Y.Y.; Chu, C. Flip-flop clustering by weighted K-means algorithm. In Proceedings of the 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016.
11. Han, K.S.; Kahng, A.B.; Li, J.J. Optimal Generalized H-Tree Topology and Buffering for High-Performance and Low-Power Clock Distribution. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 478–491. [CrossRef]
12. Wang, H.H.; Lei, Q.Q.; Yang, Y.F.; Liu, L. A Clock Tree Synthesis Scheme Based On Flexible H-tree. In Proceedings of the 2022 2nd International Conference on Electrical Engineering and Mechatronics Technology (ICEEMT), Hangzhou, China, 1–3 July 2022.
13. Mangiras, D.; Mattheakis, P.; Ribet, P.O. Soft-Clustering Driven Flip-flop Placement Targeting Clock-induced OCV. In Proceedings of the 2020 International Symposium on Physical Design, ISPD '20, Taipei, Taiwan, 20–23 September 2020.
14. Kundu, S.; Mehta, N.A.; Sircar, A. Machine Learning Based Flip-Flop Clustering for Clock Network Power Improvement. In Proceedings of the IEEE International Conference on Emerging Electronics (ICEE), Bangalore, India, 11–14 December 2022.
15. ISPD 2010 High Performance Clock Network Synthesis Contest. Available online: http://archive.sigda.org/ispd/contests/10/ispd10cns.html (accessed on 28 September 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.